# Decades of GNU Patch and Git Cherry-Pick: Can We Do Better?

Alexander Schultheiß
University of Bern
Bern, Switzerland
alexander.schultheiss@students.unibe.ch

Alexander Boll
University of Bern
Bern, Switzerland

Paul Maximilian Bittner
Ulm University and TU Braunschweig
Braunschweig, Germany

Sandra Greiner
University of Southern Denmark
Odense, Denmark

Thomas Thüm
TU Braunschweig
Braunschweig, Germany

Timo Kehrer
University of Bern
Bern, Switzerland

## Abstract

Patching is a fundamental software maintenance and evolution task enabling the (semi-)automated propagation of changes across different software versions. Established and widely used language-agnostic patchers, such as *GNU patch* and *Git cherry-pick*, work on textual artifact representations (i.e., text files) and typically rely on line numbers and contexts (i.e., surrounding unchanged text fragments) to apply changes. This strategy often fails if source and target of a patch differ, provoking cumbersome manual effort. In this paper, we study the effectiveness of commonly-used patchers, and propose a novel technique that significantly increases patch automation. First, we curate and analyze a large dataset of more than 400,000 patch scenarios (i.e., cherry picks) from 5,000 GitHub projects. Next, we examine the effectiveness of established patchers on the gathered patch scenarios. Third, we develop a novel language-agnostic patch technique, *mpatch*, that utilizes a source-to-target matching to determine suitable change locations. By comparing *mpatch* to other patchers, we find that it can correctly apply 44% more patches automatically than other language-agnostic patchers, while it also requires fewer manual fixes in cases that cannot be automated completely. Thus, *mpatch* considerably reduces the burden of manually fixing failed patches in practice, specifically in projects with frequent patch applications.

## CCS Concepts

• **Software and its engineering → Software configuration management and version control systems**; *Software maintenance tools.*

## Keywords

patching, variant synchronization, change propagation

**ACM Reference Format:**
Alexander Schultheiß, Alexander Boll, Paul Maximilian Bittner, Sandra Greiner, Thomas Thüm, and Timo Kehrer. 2026. Decades of GNU Patch and Git Cherry-Pick: Can We Do Better?. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3764537


*ICSE '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764537

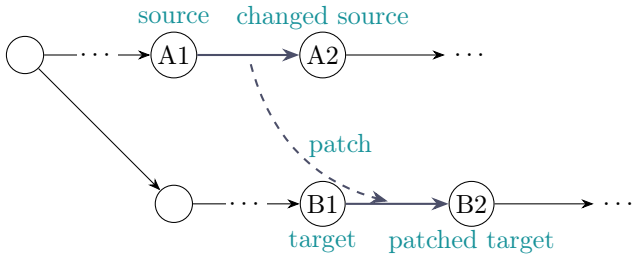


**Figure 1: Re-applying changes from a source on a target version via a patch. All names are clickable links to a patch scenario in the Apache hadoop project, which was applied in commit *28715b5*.**

## 1 Introduction

Patching is a core software maintenance activity which allows developers to propagate fine-grained software updates among software versions. Patches are applied by patching tools or *patchers*, such as *GNU patch* which is potentially the most well-known implementation of document patching. *GNU patch* lies at the heart of many software maintenance tasks for decades, such as contributing to the Linux kernel [14]. Patches summarize changes to a file that can be reapplied to copies of the file. These copies may reside, for instance, on different development branches in a version control system [8, 39]. Figure 1 presents an example of such a *patch scenario* in which a developer applies a patch from a source to a target version. Contrary to merging development branches [25], patching usually does not transfer all changes that occurred on a branch but a desired subset of changes, which is also known as cherry picking [29]. Most patchers expect a list of changes performed on a source version as input, and apply these changes to one or more target versions in form of a *patch*. For each target version, they try to identify the most suitable locations for the changes in the target and which changes should be applied.

While patching is trivial in cases where the source and target versions of the patch are identical, it becomes challenging when the source and target versions differ [35, 45]. In such cases, patchers often apply the changes at wrong locations or even fail to apply them at all. For such complex patch scenarios, prior work found that current language-agnostic patchers provoke a particularly high

rate of rejected changes in the Linux kernel [35, 45] requiring costly manual fixes.

To address the shortcomings of established language-agnostic patchers, such as *GNU patch*, several *language-specific* patchers have been proposed in the literature [23, 24, 31, 35, 36, 38, 45]. Typically, language-specific patchers parse code into structured representations, such as abstract syntax trees (ASTs), on which they then operate. This allows them to transform patches to align them with the target variant, and to complement a patch with additional changes that are required to receive a correct program (e.g., by adding missing includes). However, their applicability is limited to the patching of source code, specifically source code written in a programming language that the patcher supports. Therefore, they may complement language-agnostic patchers, but they cannot generally replace them. Even for source code artifacts, language-specific techniques proposed within the research literature in the broader context of software version control have had little to no practical impact [10]. This observation remains true today.

In this work, we acknowledge the status quo that language-agnostic, general-purpose patchers, such as GNU Patch and Git Cherry-Pick, remain the state-of-the-art in document patching in the context of software version control. However, we show that significant improvements are still possible, even without loosing the general applicability of current text-based patchers. Specifically, we significantly increase the percentage of patches that can be correctly applied without additional manual effort by proposing a novel language-agnostic patcher, called *mpatch*. During patching, *mpatch* computes a matching, identifying the commonalities of the source and target version (i.e., common lines of text). Using this *match-based* approach, it applies patches more successfully than other language-agnostic patchers without loosing applicability.

In a large-scale evaluation, we demonstrate that current language-agnostic patchers perform unsatisfactory in various patch scenarios. We mine a dataset comprising 423,717 patch scenarios from 5,000 popular projects on GitHub, covering the 10 most used project languages. We evaluate the effectiveness of the language-agnostic patchers *GNU patch*, *Git apply*, and *Git cherry-pick* by measuring precision, recall, and the number of required manual fixes. Overall, the patchers exhibit high precision but a low automation percentage. The considered patchers apply less than 50% of complex patches correctly, provoking great manual effort by the developers.

We then compare the quality of *mpatch* with the established patchers and find that *mpatch* significantly increases the number of patches that are correctly applied, reducing the number of patches that require manual fixes. Furthermore, *mpatch* requires significantly fewer fixes if it does not apply a patch correctly. To gauge the potential impact of improving language-agnostic patching, we investigate projects from our dataset that heavily use patching. We compute the impact *mpatch* would have, if the resulting projects adopted it. As a result, we find that *mpatch* has the potential for substantially reducing the manual efforts in such projects.

Overall, we gain strong empirical evidence that *mpatch* considerably improves language-agnostic patching while remaining applicable to a broad spectrum of patch scenarios in various projects. In summary, we contribute

- a dataset comprising 423,717 patch scenarios (i.e., cherry-picks) mined from 5,000 popular projects on GitHub;
- a study of the prominence of patching in practice and the frequency of challenging patch scenarios;
- *mpatch*: a novel language-agnostic patcher that utilizes a source-to-target matching;
- an extensive empirical evaluation and comparison of the effectiveness of *mpatch* and other language-agnostic patchers in a mutlitude of patch scenarios;
- an open-source reproduction package [33], comprising our novel dataset, our implementation of *mpatch*, the experimental setup, and all results.

## 2 Motivation

In this section, we first describe the concept of patching using a motivational example. We then present the state-of-the-art language-agnostic patching tools and explain their approaches.

### 2.1 Language-Agnostic Patching

Language-agnostic patching, from here on referred to as 'patching', reduces the effort of applying the same changes to multiple versions of a file. Typically, such changes are first performed manually on one version and should be repeated on other versions that may benefit from these changes, e.g., because the changes fix a crucial bug. Changes can be applied to a target version as a *patch*.

In practice, a patch typically is represented as a *diff* that documents changes to a source version, including unchanged text that surrounds the changes (i.e., their *context*) in a unified format, aka. (asymmetric) difference [16], (directed) delta [8], or edit script [15]. A *unified* diff aggregates changes with overlapping context in a *hunk* that groups related changes and defines the location of the changes in the source file. Figure 2 presents an example of such a unified diff, which we adapted from the commit ba66f3b in Apache hadoop. It comprises several changes that insert new code for a Transport Layer Security verification in the Java file `WebHdfsFileSystem.java`, grouped into two hunks (starting at lines L181 and L242, as indicated by the "`@@ ... @@`" headers). Lines to be inserted by the patch are shown in green, surrounded by three context lines.

Over time, source and target of a patch may diverge – this is the case for the source and target of our example. If a divergence affects files to be patched, a patch created from the diff between the source and changed source may no longer suit the target. The changes in the two hunks of the patch shown in Figure 2 cannot be easily applied to the target because they need to be placed in a different location. Figure 3 shows excerpts of the source and target file, focusing on the location where the Line "`private boolean isTLSKrb;`" from the first hunk should be inserted. Common lines are highlighted in blue. In the source file, the line is inserted below L183, while in the target file it should be inserted below L171.

For the second hunk of our example, we observe a similar alteration in location and context. In general, depending on how much the source and target versions diverged, some changes cannot be applied because the location of some changes may not be found, or it is questionable whether a change is actually required in the

```
--- WebHdfsFileSystem.java
+++ WebHdfsFileSystem.java
@@ -181,6 +182,7 @@
[...]
  private DFSOpsCountStatistics storageStatistics;
  private KeyProvider testProvider;
+ private boolean isTLSKrb;

  /**
   * Return the protocol scheme for the FileSystem.
@@ -242,6 +244,7 @@
   .newDefaultURLConnectionFactory(connectTimeout,
       readTimeout, conf);
  }

+ this.isTLSKrb = "HTTPS_ONLY".equals(conf.get(
     DFS_HTTP_POLICY_KEY));

  ugi = UserGroupInformation.getCurrentUser();
  this.uri = URI.create(uri.getScheme() + "://" + uri.
      getAuthority());
[...]
```

**Figure 2: Adapted patch for commit ba66f3b of hadoop.**

target. Patchers typically reject changes if it is uncertain whether or where to apply them, so that they can be applied manually.

## 2.2 Current Language-Agnostic Patchers

We consider three language-agnostic patchers in this paper: *GNU patch* [43], *Git apply* [40], and *Git cherry-pick* [41]. Naturally, this list is incomplete as other version control systems (VCS) (e.g., Mercurial or Subversion), editors, and IDEs contain their own patch utilities. In this work, we focus on patchers used by Git and Unix-based operating systems. Git, with its patchers *Git apply* and *Git cherry-pick*, is the most popular VCS [12], and *GNU patch* has great impact in open-source development (e.g., Linux kernel [14]).

*2.2.1 GNU patch.* The first release of *GNU patch* [43] was in 1985. Despite its age, it is still actively maintained, with the latest commit in February 2025 (as of February 2025), and remains a core development package of many Unix-based distributions.

*GNU patch* expects a unified diff as input, and then applies these changes based on their line number and context. For each hunk in a patch, *GNU patch* searches the hunk's context, starting at the line number specified in the hunk. Searching above and below that line number, *GNU patch* applies the changes at the first location with an identical context. If it cannot find the context in the file, it relaxes the context by removing the outermost line of the leading and trailing context and repeats the search. In its default configuration, *GNU patch* repeats this context relaxation at most twice. Thus, *GNU patch* allows for differences between the source and the target of a patch, except for the most direct neighboring context. If *GNU patch* finds no suitable context, it reports the hunk as rejected.

Following this approach, *GNU patch* might apply changes incorrectly or reject required changes if the source and target versions diverged substantially. When applying the patch of Figure 2 to the target in Figure 3 (right), *GNU patch* rejects the first hunk containing the insertion of `private boolean isTLSKrb;` because the context line above the change differs. In general, *GNU patch* may reject changes if the first leading or trailing context line diverges. Besides rejecting the first hunk of our example, *GNU patch* applies the change in the second hunk to a wrong location. This mistake occurs because *GNU patch* applies changes to the first suitable location, which may not be the correct one, especially in cases with several similar contexts.

*2.2.2 Git apply.* Git comprises two patch utilities, *Git apply* [40] and *Git cherry-pick* [41]. The former is Git's counterpart to *GNU patch* and uses a similar, context-based approach. By contrast to *GNU patch*, however, *Git apply*'s default configuration is much more conservative regarding differences between the source and target version. The complete context of a hunk must fit and if a single hunk of a patch is rejected, *Git apply* rejects the entire patch. To override this behavior, *Git apply* provides a `--reject` option. This option applies all applicable hunks and reports rejected hunks in a manner similar to *GNU patch*.

In both configurations, *Git apply* rejects the two hunks of our motivating example (cf. Figure 2), as the contexts of the hunks do not exactly fit the target version. In general, we expect that *Git apply* will reject required changes more frequently than *GNU patch*.

*2.2.3 Git cherry-pick.* Git's second utility is *Git cherry-pick* [41] which can be used to transfer one or more commits between branches without a full merge of these branches. It is tightly integrated with Git's version control and requires a commit history; specifically, it can only be applied if the source and target version have a common ancestor in the commit history. Given a list of commits, *Git cherry-pick* reapplies the changes of these commits to the current working tree one-by-one, creating a new commit for each applied patch.

If source and target of a cherry pick have diverged, *Git cherry-pick* may encounter conflicts. *Git cherry-pick* tries to *merge* these files using the same merge strategy as `git merge`. Using Git's default configuration, *Git cherry-pick* applies changes if they are not part of a conflicting hunk (i.e., hunks whose context diverged in source and target). For conflicting hunks, *Git cherry-pick* writes the conflict directly into the file by concatenating the hunk of the source and the hunk of the target, highlighting it with conflict markers. Such conflicts must be resolved manually.

In our experience, *Git cherry-pick* can reliably identify the correct location for a change in most cases by considering the common ancestor of the source version and the target version. However, for diverged versions, *Git cherry-pick* may report merge conflicts even though the conflicting changes could simply be applied. Each conflict requires manual effort to resolve. For the patch of our motivating example, *Git cherry-pick* would report merge conflicts for both hunks, provoking manual intervention.

# 3 The Novel Language-Agnostic Patcher *mpatch*

Based on our practical experiences with patching, we find that the shortcomings of current language-agnostic patchers can likely be mitigated by utilizing a source-to-target matching, identifying the commonalities (i.e., common lines of text) of the versions. Current patchers either rely on a context which only contains information about the source version, but not the target, or on merging changes with respect to a common ancestor version, which may produce

```
179  private String restCsrfCustomHeader;
180  private Set<String> restCsrfMethodsToIgnore;
181
182  private DFSOpsCountStatistics statistics;
183  private KeyProvider testProvider;
184
185  /**
186  * Return the protocol for the FileSystem
187   *
188   * @return <code>webhdfs</code>
```

```
167  private Set<String> restCsrfMethodsToIgnore;
168  private static final ObjectReader READER =
169      new ObjectMapper().reader(Map.class);
170
171  private DFSOpsCountStatistics statistics;
172
173  /**
174  * Return the protocol for the FileSystem
175  * <p/>
176  *
```

**Figure 3: Excerpt from the source file (left) and target file (right) of our exemplary patch scenario from commit ba66f3b in Apache hadoop. Common lines are highlighted in blue.**

tedious merge conflicts. By contrast, a matching helps with identifying the correct locations for changes in a patch, because it explicitly identifies the commonalities between the source and target versions. To this end, we developed *mpatch*, a new match-based patcher.

## 3.1 Patching Algorithm

The main workflow of the patching algorithm used by *mpatch* is given in Algorithm 1. The algorithm relies on three configurable

**Algorithm 1** mpatch

```
1:  mpatch(V_S, V_T, Δ_{V_S⇒V_S'})                    Source, target, patch
2:  V'_T ← V_T, R ← {}                                Initialize output
3:  M_{S,T} ← match(V_S, V_T)                         Match source and target
4:  for δ ∈ Δ_{V_S⇒V_S'}                              For each change group do
5:      (δ', R') ← filter(δ, M_{S,T})                 filter invalid changes
6:      R ← R ∪ R'                                    Update rejects
7:      for d ∈ δ'                                    Apply each change do
8:          l_S ← select l ∈ V_S changed by d
9:          l_T ← select l ∈ V_T with (l_S, l_T) ∈ M_{S,T}
10:         V'_T ← apply(V'_T, M_{S,T}, l_T, d)
11:     end for
12: end for
13: return (V'_T, R)                                  patched target and rejected changes
```

operators, match, filter, and apply, which we will explain later in this section.

The input for the algorithm is the source version $V_S$, the target version $V_T$, and a patch $\Delta_{V_S \Rightarrow V_{S'}}$ (aka. directed delta). A version $V$ is the set of all lines of text in all files associated with the version, and each line is equipped with a line number and the respective file paths. Each change group $\delta \in \Delta_{V_S \Rightarrow V_{S'}}$ consists of related changes and their corresponding lines (e.g., $\delta$ could be a hunk of a unified diff). First, *mpatch* initializes the patch result $V'_T$ and the set of rejected changes $\mathcal{R}$. Then, the match operator determines a matching $M_{S,T} \subseteq V_S \times V_T$ between $V_S$ and $V_T$ that defines the corresponding lines of the source and target versions.

Starting from line 4, *mpatch* iterates over each change group $\delta \in \Delta_{V_S \Rightarrow V_{S'}}$ and filters invalid changes with the filter operator, yielding a filtered change group $\delta'$ and a set of filtered (aka. rejected) changes $\mathcal{R}'$. Whether a change is invalid is determined based on the matching $M_{S,T}$. For example, a change is invalid if it removes a line that does not exist in the target variant. Without a filter, *mpatch* would still be able to apply changes, but it would not be able to determine whether lines inserted in the source version should also be inserted in the target version, which could lead to incorrect insertions. After filtering, *mpatch* applies the remaining changes in $\delta'$ with the apply operator one-by-one. For each change $d \in \delta'$, *mpatch* selects the line $l_T \in V_T$ that corresponds to the changed line $l_S \in V_S$ according to $M_{S,T}$. Then, *mpatch* applies the change with the apply operator, yielding an updated target version $V'_T$. Finally, *mpatch* returns $V'_T$ and $\mathcal{R}$.

## 3.2 Implementation

Algorithm 1 requires a concrete implementation of match, filter, and apply. In the following, we define a concrete implementation of each operator, which we use to investigate the effectiveness of match-based patching with *mpatch*. With these operators, we implemented a prototype of *mpatch* [33] in Rust to evaluate *mpatch*'s effectiveness and compare it to other patchers. Our Rust implementation offers a command line interface, similar to *GNU patch*: The prototype accepts unified diffs as input for $\Delta_{V_S \Rightarrow V_{S'}}$ in which changes are grouped in hunks and consist of line insertions and deletions. The source and target version are given by the path to their root directory. With this interface, *mpatch* could already be used in most cases in which *GNU patch* is applied and could also be integrated in a VCS like Git.

*Match.* Our implementation of the match operator determines a source-to-target matching $M_{S,T}$ based on the largest common subsequence (LCS) [27] of lines between each file and its counterpart in the target version. First, *mpatch* matches the common files of source and target. Similar to *GNU patch*, *mpatch* locates the counterparts of files in the target version based on the file paths of changed files specified in the patch file. If a file has no counterpart, its content is treated as unmatched. If a counterpart is found, the source and target files are compared using the LCS to match their content. Given two files, *mpatch* applies Myers' LCS algorithm [27] that determines the LCS by iteratively searching for the shortest path in an edit graph that models all possible ways to transform one file into the other, which implicitly yields commonalities. Thereby, *mpatch* identifies the common lines of the *source* and the *target* versions, and matches the lines by creating a mapping of their locations accordingly. The common lines may include lines that are to be deleted, and lines from the context of the changes in a patch. Lines that are inserted by the patch do not exist yet and are thus not

included in the matching. There may be at most one match for each line. For the source and target files of our motivating example, LCS correctly matches the common lines of both files as highlighted in Figure 3. We selected LCS for its broad applicability.

*Filter*. Our implementation of the filter operator considers a change as undesired when it affects lines of the source version that have no match in the target. Here, *mpatch* differentiates between inserted and deleted lines. Inserted lines cannot have a match because they do not exist yet, thus, *mpatch* considers the lines above and below the inserted line in the source version. If the neighboring lines *directly* above *or* below have a match in the target, the change is kept; thereby *mpatch* accounts for prepending or appending lines to matched content. This means that an insertion is filtered if neither the context line above nor below the insertion have a matching line in the target version. In our motivating example, *mpatch* keeps the changes because they insert lines next to matched lines. Specifically, the insertion in the first hunk happens directly above several matched lines as shown in Figure 3. By contrast, deleted lines must have a match in the target, because they could not be deleted otherwise. Thus, a deletion is filtered if the to-be-deleted line has no match. Besides yielding a filtered change group $\delta'$, the filter also yields a set of filtered changes $\mathcal{R}$, i.e., rejected changes – similar to rejects being reported by *GNU patch*. These filtered changes may be reviewed by a user after patch application, and may have to be applied manually.

*Apply*. The changes in the patch are applied to the target version according to the source-to-target matching $M_{S,T}$. Our implementation of apply also differentiates between changes that insert and delete lines. For a line deletion, the line to be removed from the target is matched to the deleted source line. For lines to be inserted, however, there is no direct match that determines the location of the insertion. Therefore, *mpatch* considers the context lines adjacent to the inserted line. According to the employed filter heuristic, at least one of the context lines of an insertion must exist in both the source version and the target version. Based on the matches for the context lines, *mpatch* determines whether to apply an insertion below, above, or in between the respective context lines in the target version. For the first change of our motivating example, *mpatch* inserts the line directly below L171 in Figure 3, as this location is surrounded by the matches of the lines where the change was originally applied in the source version.

## 4 Evaluation Methodology

The central goal of our paper is to evaluate the effectiveness of different approaches to language-agnostic patching, and improve patch automation in scenarios where the target of a patch differs from the source. To this end, we now present our study that aims to understand the effectiveness of current patchers in such scenarios (**RQ1**), to which degree our new patcher *mpatch*, presented in the prior section, can improve patch automation (**RQ2**), as well as the potential impact of match-based patching (**RQ3**).

**RQ1**: *How effective are existing language-agnostic patchers in complex patch scenarios?*

Previous work on patching in the Linux kernel found that *GNU patch* may fail to patch C source code correctly, once patches become more complex; i.e., if the location for changes differs in the target [35]. However, it is unclear whether these observations generalize to patch scenarios in other projects, and whether other patchers exhibit the same limitations or automation potential.

**RQ2**: *How effective is mpatch in comparison to existing language-agnostic patchers?*

As a result of RQ1, we observe that existing patchers frequently fail to identify the correct location to apply a change. By manually inspecting failing cases, we find that matching the source and target of a patch application might solve most of the encountered problems. Consequently, we developed *mpatch* (cf. Section 3), a novel language-agnostic patcher that relies on a matching.

**RQ3**: *What is the potential impact of improving language-agnostic patching with mpatch?*

To assess the significance of patching in practice and to gauge the impact of *mpatch*'s potential improvement over existing patchers, we study to which extent patching is a relevant software maintenance activity in practice. Specifically, patching may be more prevalent in certain types of communities or repositories (e.g., depending on the main programming language), which then would also be more heavily burdened by failing patches.

### 4.1 Data Collection

To answer our research questions, we collected a dataset of patching scenarios, specifically cherry picks (i.e., patches applied with *Git cherry-pick*), mined from public repositories. First, we curated a set of 5,000 public projects from GitHub using a similar methodology as Boll et al. [6] who studied merge conflicts in public projects. Following their approach, we selected the ten most popular project languages on GitHub in the first quarter of 2024 by means of their number of stars. We identified the most popular languages based on the language overview of the GitHut project [3]. The most popular languages on GitHub were: Python, JavaScript, Go, C++, Java, TypeScript, C, C#, PHP, and Rust. Next, we collected 500 projects from the most popular projects for each language using GitHub's REST API: We filtered projects by language and sorted them by their number of stars in descending order, then selecting the top results. To identify a project's (programming) language, GitHub selects the language of the majority of artifacts in the repository, though some project artifacts are often written in another language (e.g., shell scripts, configuration files, documentation, or build files).

Within our dataset, we identified cherry picks among all commits and branches of the projects. While Git provides a dedicated `Git cherry` command that is meant to find cherry picks by comparing the diffs of commits, this command does not work for cases in which the patch performed by a cherry pick differs in the source and target – which are exactly the cases that are interesting to us. Fortunately, users can configure *Git cherry-pick* to append a line to the commit message that states "*(cherry picked from commit <id>)*" using the "`-x`" flag. To identify cherry picks, we parse the commit messages of all commits in a repository, searching for instances of such a line. Using the specified *<id>* of the commit, we can then retrieve the commit. In a few repositories that rewrote their Git histories (using `Git rebase`), we could not find some picked commits and thus could not include them in our evaluation.

## 4.2 Evaluation of Patcher Effectiveness

*4.2.1 Considered Patchers.* We consider *GNU patch*, *Git apply*, and *Git cherry-pick* as state-of-the-art for language-agnostic patching (cf. Section 2.2). We use the default configurations for *GNU patch*, and *Git cherry-pick* because they are likely the most-used configurations in practice. To not discriminate against *Git apply* in our evaluation, we invoke it with the `reject` flag (cf. Section 2.2.2), which simulates a reject behavior similar to *GNU patch*, instead of aborting patching.

*4.2.2 Sampling of Patch Scenarios.* Due to the large size of our dataset with more than 400,000 cherry picks (cf. Table 2), we focus our analysis on patch scenarios for which we expect differences between patching techniques. To this end, we classify cherry picks into the classes 'trivial' and 'complex'. We classify a patch as trivial iff the diff between the source and changed source and the diff between the target and patched target (cf. Figure 1) are identical, down to each line number and whitespace. Trivial cherry picks do not require adjustments when being applied to the target version, and can be applied solely based on the line numbers of changes. In fact, for trivial patches, we observed that all patchers, including *mpatch*, performed perfectly in essentially all cases, with only rare cases causing issues (e.g., intentionally broken character encodings in files). Hence, a new patching technique cannot improve the outcome for trivial patches. We discard trivial patches from further analysis, and instead focus on the remaining 101,196 (23.6%) complex patches to answer our research questions.

*4.2.3 Automated Application of Patchers.* To assess the effectiveness of a patcher, we replay each patch scenario in our dataset. As illustrated in Figure 1, a patch scenario consists of

- a patch that we extract from the diff of source *A1* and changed source *A2* (i.e., the cherry),
- a target version *B1*, that is the commit upon which the *Git cherry-pick* command was originally applied,
- and the expected patched target *B2*, which corresponds to the commit after the cherry pick, i.e., the ground truth.

To replay a patch scenario, we generate source, cherry, and target versions of the patch scenario, and we invoke a patcher with its command line interface. We then compare the patch outcome to the expected patched target (see Section 4.2.4).

An additional step is required for *Git cherry-pick*. By contrast to the other patchers, *Git cherry-pick* does not directly reject changes but instead reports merge conflicts (cf. Section 2.2.3) that are written directly into the file. To appropriately compare *Git cherry-pick* to the other patchers, we thus had to handle its conflicts. For comparability, we choose an approach that reflects the behavior of the other patches as closely as possible. Other patchers write their rejected changes into a *rejects* file, instead of applying them. We mimicked this behavior for *Git cherry-pick* by removing all Git-markers and the *theirs* versions from a file after *Git cherry-pick* finished. For conflicting locations, this left only the original content (the *ours* version), effectively simulating a rejection of conflicting hunks.

*4.2.4 Classification of Patch Outcomes.* We classified the outcome of a patch similarly to the methodology of Schultheiß et al. [32]. All our patch scenarios stem from real-world version histories. Thus, we can determine whether a patch has been applied correctly by comparing the result with how developers committed a patch application in their project's history. First, we compare the patched target version with the expected patch outcome using Unix `diff`, where we treat missing files as empty and ignore trailing whitespace, as it rarely carries important information. According to the observed differences in the patched target, we then classify the changes in a patch as exactly one of the five classes shown in Table 1. The classes distinguish required changes that must be applied by a patcher, and undesired changes that must not be applied and hence rejected (first column). For required changes, there are three cases that may occur (second to last column): the change has been applied to the correct location (*correct*), the change has been applied to an incorrect location (*wrong location*), and the change has not been applied (*missing*). For undesired changes, we also distinguish between a change having been applied (*invalid*) anywhere, and a change not having been applied (*filtered/rejected*).

**Table 1: Overview of patch outcome classification**

| Change Type | Applied? | Location Correct? | #Observed Differences | Patch Outcome | Class |
|---|---|---|---|---|---|
| Required | ✓ | ✓ | 0 | *correct* | TP |
| Required | ✓ | × | 2 | *wrong location* | FP & FN |
| Required | × | N/A | 1 | *missing* | FN |
| Undesired | ✓ | N/A | 1 | *invalid* | FP |
| Undesired | × | N/A | 0 | *filtered/rejected* | TN |

Using this methodology, we focus only on the patch application itself, effectively ignoring differences that are not directly related to the content of the patch. We ignore these differences because the evaluated language-agnostic patchers focus on applying patches, and do not alter the target in any other way. Furthermore, we only consider text files in our evaluation – excluding binary files which are difficult to evaluate: A small edit in a binary file can be just as problematic as a large one, and a diff cannot effectively represent differences in binary files.

*4.2.5 Evaluation Metrics.* We measure effectiveness of patchers in terms of six different metrics.

*Patch Automation Percentage and Required Fixes.* These metrics reflect the manual effort of developers when patching. The automation percentage measures how often a patcher is able to apply a patch without human intervention. We determine the patch automation percentage as the ratio of the number of patches with 100% precision and recall (i.e., no fixes needed), and the total number of patches. We also measure the average number of fixes that are required after a patch application by counting the number of lines that have to be changed to correct the artifact.

*Precision, Recall, and F1-Score.* We measure precision and recall because they reflect whether required changes are correctly applied and whether undesired changes were rejected, and F1-Score as it reflects the trade-off between precision and recall. To determine precision and recall, we consider patchers as classifiers for changes [35] that determine whether a change is *required* or *undesired*, with required changes being positive and undesired changes negative instances. Thus, for required changes, we count correctly

applied changes as true positive (TP), and rejected changes as false negatives (FN) (cf. Table 1). For undesired changes, we count applied changes as false positive (FP), and not-applied changes as true negative (TN). Required changes that were applied to the wrong location represent a special case because they cause an undesired change (at the wrong location) and a missing change (at the correct location); we counted them as both FP and FN, which reflects the additional effort of developers in such cases.

Precision is the ratio of correctly applied required changes among all applied changes, which is given by $\frac{TP}{TP+FP}$, while recall is the ratio of required changes that were applied correctly, which is given by $\frac{TP}{TP+FN}$. Lastly, the F1-Score is given by $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$.

*Patch Application Time in Seconds.* Lastly, we measure the time a patcher requires to apply a patch. The measured time includes all steps related to the application of a patch, such as matching source and target version, identifying which lines of a file should be changed, applying the changes, saving changed files, and reporting failed changes. The measurement does not include setup and clean-up tasks of the evaluation, such as the preparation of the source and target versions, or the calculation of our evaluation metrics.

# 5 Results

## 5.1 Overview of Mined Cherry Picks

In Table 2, we present an overview of our gathered dataset with which we examine the prominence of patching in open source development. In total, we mined cherry-picks from 5,000 repositories (500 from each of the 10 most popular languages on GitHub). While in 4,304 projects we cannot find cherry picks with our commit message analysis, we still find a total of 423,717 cherry picks spread across 696 repositories. The column 'cherry pick [%]' denotes the cherry-pick-to-commit ratio; i.e., how many percent of the commits are cherry picks. Next, the column 'complex cherry pick [%]' indicates the percentage of cherry picks that are complex, meaning the source and target do not match perfectly. The complex cherry picks represent the patching scenarios we studied for RQs 1 and 2.

Language-wise, we observe notable differences, specifically in the absolute and relative occurrence of cherry picks. The projects implemented mainly in C and C++ encompass half of all identified cherry picks in absolute numbers whereas JavaScript projects use them least frequently in absolute and relative numbers. Conversely, Java projects exhibit the highest cherry-pick-to-commit ratio on average: more than every 50$^{\text{th}}$ commit is a cherry pick. Moreover, we observe differences regarding the ratio of complex cherry picks among the identified cherry picks. Projects implemented in PHP exhibit the lowest ratio of complex cherry picks (on average: 13.2%), while C++ and Rust projects have the highest ratio with averages of 31.8% and 31.9%, respectively. The average ratio of complex cherry picks across all projects is 23.9%. Thus, almost every fourth cherry pick requires target-specific adjustments (e.g., applying changes to locations different from the source).

## 5.2 Evaluation of Patchers on our Dataset

To answer RQs 1 and 2, we studied the patchers introduced in Section 2.2 and *mpatch* on our dataset. Table 3 presents the key results of our evaluation. For each patcher, we measured precision, recall, automation degree, the number of required fixes for a failed patch, and its execution time (cf. Section 4.2.5). Values highlighted in bold font perform best for each metric and project language. The rightmost columns present the mean ($\bar{x}$) over all languages, the relative difference of these means of the patchers compared to *mpatch* (±%), and the effect size $|r_{RB}|$, computed with the rank-biserial correlation [9]. To test the null hypothesis whether the results of *mpatch* or of another patcher stem from the same distribution, we used the Wilcoxon signed-rank test [44]. In all cases, the hypothesis got rejected with $p \ll 0.01$. This shows that *mpatch* behaves significantly different than the other patchers for all metrics measured.

**Table 2: Overview of our collection of GitHub projects.**

| main repository language | #sampled projects | #projects w. cherry picks | #cherry picks | cherry pick [%] | complex cherry pick [%] |
|---|---|---|---|---|---|
| C | 500 | 74 | 108,595 | 1.292 | 19.1 |
| C# | 500 | 61 | 6,929 | 0.334 | 22.1 |
| C++ | 500 | 112 | 108,085 | 1.437 | 31.8 |
| Go | 500 | 98 | 28,941 | 1.234 | 28.6 |
| Java | 500 | 91 | 81,859 | 2.232 | 20.6 |
| JavaScript | 500 | 43 | 3,243 | 0.187 | 22.4 |
| PHP | 500 | 54 | 22,593 | 0.830 | 13.2 |
| Python | 500 | 57 | 33,152 | 1.130 | 27.1 |
| Rust | 500 | 39 | 7,033 | 0.329 | 31.9 |
| TypeScript | 500 | 67 | 23,287 | 0.728 | 19.2 |
| total | 5,000 | 696 | 423,717 | 1.153 | 23.9 |

Next, we describe the results presented in Table 3 in detail, discussing the metrics and analyzing language-specific results.

*Patch Automation Percentage: mpatch* achieves the greatest automation percentage across all project languages and is able to correctly apply 59% of complex patches automatically, on average. In comparison, *GNU patch* correctly applies 40% of patches, *Git apply* 16%, and *Git cherry-pick* 41%.

*Required Fixes:* The number of lines that must be fixed by developers after patching shows variance between project languages, with patches in PHP projects requiring the fewest fixes and patches in C# projects requiring the most. *mpatch* performs best across all project languages with 18 fixes needed on average. Current patchers require noticeable more fixes on average, ranging from 26 fixes (*Git cherry-pick*) to 46 fixes (*Git apply*). On average, current patchers require 47% to 155% more manual fixes after being used.

*F1-Score: mpatch* achieves the best F1-score across all project languages with an average of 0.95, and therefore the best trade-off between precision and recall. In comparison, *GNU patch* and *Git cherry-pick* achieve an F1-score of 0.9 and *Git apply* performs the worst with an average F1-score of 0.84.

*Precision:* All patchers exhibit high precision (> 0.9), with *Git apply* showing the highest precision for every project language. *mpatch* performs (slightly) worse than current patchers, which achieve a 1.5% to 2.7% higher precision at the cost of reduced recall.

**Table 3: Comparison of the considered patchers for various project languages.**

| Metric | Patcher | Project Languages | | | | | | | | | | $\bar{x}$ | ±% | $\lvert r_{RB}\rvert$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Python | JS | Go | C++ | Java | TS | C | C# | PHP | Rust | | | |
| Autom. (%) | *mpatch* | **59.93** | **62.09** | **61.78** | **53.26** | **59.92** | **57.93** | **66.04** | **49.50** | **56.47** | **50.07** | **59.11** | | |
| | *GNU patch* | 44.18 | 39.25 | 35.38 | 36.64 | 40.09 | 40.62 | 45.30 | 33.74 | 45.77 | 34.75 | 40.29 | −31.83% | 0.36 |
| | *Git apply* | 24.22 | 14.65 | 13.85 | 16.38 | 11.72 | 20.14 | 12.94 | 15.30 | 26.81 | 16.56 | 15.78 | −73.30% | 0.77 |
| | *Git cp* | 42.03 | 36.73 | 48.85 | 37.08 | 38.78 | 42.28 | 44.18 | 33.81 | 44.64 | 36.17 | 41.00 | −30.60% | 0.35 |
| Req. Fixes | *mpatch* | **11.53** | **8.78** | **27.03** | **23.50** | **8.90** | **12.04** | **18.91** | **69.59** | **5.06** | **10.82** | **17.92** | | |
| | *GNU patch* | 37.17 | 90.74 | 53.66 | 51.95 | 33.38 | 35.43 | 30.39 | 136.17 | 15.56 | 37.45 | 41.57 | 132.03% | 0.37 |
| | *Git apply* | 40.22 | 93.60 | 58.14 | 56.39 | 38.59 | 38.46 | 33.84 | 139.19 | 17.70 | 44.57 | 45.69 | 154.95% | 0.56 |
| | *Git cp* | 23.48 | 81.85 | 35.99 | 30.87 | 22.39 | 20.92 | 19.87 | 72.35 | 10.19 | 26.77 | 26.34 | 46.90% | 0.32 |
| F1 Score | *mpatch* | **0.94** | **0.95** | **0.96** | **0.94** | **0.96** | **0.96** | **0.95** | **0.93** | **0.93** | **0.96** | **0.95** | | |
| | *GNU patch* | 0.89 | 0.88 | 0.92 | 0.87 | 0.92 | 0.91 | 0.90 | 0.86 | 0.90 | 0.90 | 0.90 | −5.51% | 0.34 |
| | *Git apply* | 0.84 | 0.82 | 0.88 | 0.82 | 0.86 | 0.88 | 0.81 | 0.82 | 0.87 | 0.86 | 0.84 | −11.44% | 0.59 |
| | *Git cp* | 0.90 | 0.90 | 0.93 | 0.89 | 0.92 | 0.93 | 0.90 | 0.88 | 0.90 | 0.92 | 0.90 | −4.66% | 0.32 |
| Precision | *mpatch* | 0.94 | 0.92 | 0.96 | 0.92 | 0.96 | 0.95 | 0.94 | 0.91 | 0.90 | 0.95 | 0.94 | | |
| | *GNU patch* | 0.96 | 0.93 | 0.97 | 0.93 | 0.97 | 0.97 | 0.95 | 0.93 | 0.92 | 0.96 | 0.95 | 1.50% | 0.19 |
| | *Git apply* | **0.97** | **0.98** | **0.98** | **0.95** | **0.98** | **0.98** | **0.96** | **0.96** | **0.92** | **0.97** | **0.96** | 2.62% | 0.32 |
| | *Git cp* | 0.95 | 0.95 | 0.97 | 0.94 | 0.97 | 0.97 | 0.95 | 0.94 | 0.91 | 0.96 | 0.95 | 1.58% | 0.18 |
| Recall | *mpatch* | **0.95** | **0.94** | **0.97** | **0.96** | **0.97** | **0.96** | **0.96** | **0.92** | **0.98** | **0.97** | **0.96** | | |
| | *GNU patch* | 0.81 | 0.75 | 0.84 | 0.77 | 0.84 | 0.80 | 0.77 | 0.76 | 0.87 | 0.84 | 0.80 | −16.78% | 0.70 |
| | *Git apply* | 0.70 | 0.59 | 0.75 | 0.64 | 0.70 | 0.71 | 0.55 | 0.62 | 0.77 | 0.74 | 0.66 | −31.76% | 0.86 |
| | *Git cp* | 0.84 | 0.77 | 0.88 | 0.81 | 0.85 | 0.85 | 0.80 | 0.78 | 0.89 | 0.87 | 0.83 | −13.44% | 0.66 |
| Time (s) | *mpatch* | 0.03 | 0.03 | 0.14 | 0.12 | 0.04 | 0.16 | 0.03 | 0.03 | 0.03 | 0.04 | 0.07 | | |
| | *GNU patch* | **0.01** | **0.01** | **0.01** | **0.01** | **0.01** | **0.01** | **0.01** | **0.01** | **0.02** | **0.01** | **0.01** | −83.09% | 0.47 |
| | *Git apply* | 0.13 | 0.05 | 0.18 | 0.42 | 0.31 | 0.26 | 0.23 | 0.16 | 0.09 | 0.07 | 0.27 | 266.32% | 0.90 |
| | *Git cp* | 0.07 | 0.05 | 0.07 | 0.26 | 0.14 | 0.12 | 0.12 | 0.09 | 0.06 | 0.05 | 0.14 | 96.09% | 0.82 |

*Recall:* On average, *GNU patch* and *Git cherry-pick* achieve recalls of 0.80 and 0.83, respectively; *Git apply* performs the worst with a recall of 0.66. By contrast, *mpatch* achieves an average recall of 0.96 and it outperforms the remaining patchers across all project languages. On average, we observe that the best current patcher, *Git cherry-pick*, has a 13.44% lower recall than *mpatch*.

*Patch Application Time:* In terms of average patch application time, we observe no outstanding differences between patchers for the vast majority of patches. We find that all patchers need much less than one second per patch on average and that the *runtime differences* between the patchers are smaller than one second for more than 99.00 % of patches. While *GNU patch* is the fastest patcher with an average patch time of 0.01 s, *mpatch* with a patch time of 0.07 s is also faster than *Git apply* (0.27 s) and *Git cherry-pick* (0.14 s) for more than 95.00 % of patches. We observe very few (< 0.02 %) outliers for which *mpatch* (maximum 106.00 s) and *Git apply* (maximum 130.00 s) exhibit a considerably longer patch time of about one to two minutes. *GNU patch* (maximum 3.00 s) and *Git cherry-pick* (maximum 14.00 s) are not affected by outliers and present no extraordinarily long patch times.

Language-wise, we observe some variance; e.g., all patchers have lower precision and recall for C# projects. Similarly for automation level, we observe the worst performance across all patchers for C#. Moreover, we find that patch scenarios in PHP projects require fewer fixes than scenarios in other project languages. The number of required fixes varies largely w.r.t. the project languages and patchers; e.g., for JavaScript, patches applied by *mpatch* need one-tenth of the fixes compared to the remaining patchers.

All in all, *mpatch* considerably improves the degree of automation, the number of required fixes, and the recall in comparison to current patchers, at the cost of a small reduction in precision.

## 5.3 Impact on Frequently Patched Projects.

For RQ3, we compare the performance of *mpatch* to the currently best-performing patcher *Git cherry-pick* in the five projects with the highest number of cherry picks. Table 4 presents the result of this comparison. The ratio of cherry picks to commits ('cherry pick [%]') shows that the projects utilize cherry picking very differently. Some projects mostly commit trivial cherry picks for which we do not expect differences between the patchers. For instance, 'intellij-community' features only 859 complex cherry picks out of 12,998 cherry picks in total. Next, the table presents the amount of required fixes on average per complex patch (columns 'required fixes <tool>'). We can then calculate the total number of required *manual* fixes in a project; e.g., for 'ceph' this would be $30.8 \frac{\text{fixes}}{\text{complex cherry picks}} \cdot 7{,}758$ complex cherry picks = 238,946 fixes for *Git cherry-pick* and $18.0 \cdot 7{,}758 = 139{,}644$ for *mpatch*. The last

**Table 4: Comparison of automation potential of *mpatch* and *Git cherry-pick* in projects with the most cherry picks.**

| repository | cherry pick [%] | #cherry picks | #complex cherry picks | #required fixes *Git cp* | #required fixes *mpatch* | fully autom. *Git cp* | fully autom. *mpatch* |
|---|---|---|---|---|---|---|---|
| IntelliJ | 2.45 | 12,998 | 859 | 6.8 | 3.0 | 56.7% | 66.0% |
| Hadoop | 21.02 | 14,553 | 3,618 | 15.6 | 5.0 | 34.2% | 62.0% |
| FreeBSD | 2.31 | 21,266 | 3,316 | 8.3 | 4.3 | 42.3% | 67.0% |
| FFmpeg | 17.99 | 23,774 | 3,829 | 3.3 | 1.2 | 55.0% | 76.4% |
| ceph | 22.01 | 32,651 | 7,758 | 30.8 | 18.0 | 36.0% | 59.3% |

two columns show the automation percentage, which represents the percentage of patches that were applied correctly by a patcher.

# 6 Discussion

## 6.1 Answers to Our Research Questions

***RQ1**: Effectiveness of current general-purpose patchers.* In our evaluation of current patchers, *Git cherry-pick* performed best. Compared to *GNU patch*, it has the advantage that it is integrated directly into a version control system. This saves manual effort to commit changes, especially for trivial patches that can be applied without fixes. However, it comes at the price of *Git cherry-pick* being less applicable, specifically to Git projects only. For instance, it is difficult to create a patch and transfer it to another project. Moreover, *Git cherry-pick* does not outperform *GNU patch* by a great margin as its merge-based strategy is susceptible to any change in the context being reported as conflict. In our evaluation, we inspected several patches manually, to understand the workflow of patchers and their strengths and weaknesses. We observed cases in which *Git cherry-pick* reported extremely complex merge conflicts, even if the patch only contained one or two changes. For example, the cherry pick applied as commit 0539294 in the project moby changed a single line in the target, but upon replaying, Git reports a merge conflict spanning several hundred lines. Such extreme cases highlight the limitations of the merge-based strategy.

In general, current patchers achieve high precision but fall short in the recall and patch automation rates. Overall, less than half of the patches can be automatically applied with current patchers. In terms of recall and automation, *Git apply* performed much worse than *GNU patch* and *Git cherry-pick*. This is likely due to its strict rejection heuristic: any difference in the target's context leads to a rejection (cf. Section 2.2.2). Interestingly, this behavior only increases precision marginally. By comparing the results of *Git apply* and *GNU patch*, we find that a partially common context is a reliable indicator that a change is required as *GNU patch* achieves a much higher recall with its more relaxed context-based strategy.

All examined patchers apply patches very fast in most cases – *GNU patch* being the fastest patcher and *Git apply* the slowest on average. *Git apply* is the only patcher exhibiting outliers among the current patchers. We find that it is especially affected by patches with a large number of changes to many different files. For instance, *Git apply*'s strongest outlier (130s) is a huge patch in the k3s project that affects 386 files and contains more than 500,000 changes. Here, the other patchers required only about one second for the patch application. However, we expect minimal practical impact of the runtime differences, as almost all patches can be applied in less than one second by any patcher.

> ***RQ1:** Git cherry-pick is the most effective current patcher, when Git histories are available. While current patchers can identify undesired changes reliably, they often reject required changes or apply them at wrong locations. To increase automation, new patch tools should improve recall as this promises better overall results.*

***RQ2**: Effectiveness of mpatch.* Our results show that *mpatch* performs consistently better than current patchers across all project languages. Specifically, the automation percentage and number of required fixes are improved considerably: *mpatch* can apply 44% more patches correctly than *Git cherry-pick* (i.e., automation percentages of 59% vs. 41%), and *mpatch* requires 8.4 fewer fixes per patch on average. We observe a similar improvement for the Top-5 projects with the highest number of patches (cf. Table 4).

The improvement in automation percentage and required fixes is also reflected by an improved F1-score, which represents the trade-off between precision and recall. While the precision of *mpatch* is (slightly) lower across all project languages, its recall is considerably higher. This is most likely due to its match-based approach being able to identify the correct locations for required changes more often than a context-based approach. By matching entire files, *mpatch* applies fewer changes to similar, but incorrect locations and rejects fewer (required) changes for which the context differs between the source and the target.

In terms of patch application time, *mpatch* applies patches faster than *Git apply* and *Git cherry-pick* for 95% of patches, while *GNU patch* is still slightly faster on average. Similar to *Git apply*, *mpatch* also exhibits very few ($< 0.017\%$) outliers that require a considerably longer time to patch. We find that these outliers are patches that change extremely large files, which require a long time to match. For instance, we observed the longest patch time (106s) for a patch to a file with more than 90,000 lines of text in the Elastic kibana project. Nevertheless, such cases are rare and *mpatch* is able to apply almost all patches at a similar speed compared to other patchers.

Overall, our prototypical implementation of *mpatch* applies patches more successfully than all other examined patchers, including *Git cherry-pick*, despite biases in favor of *Git cherry-pick* (cf. Section 6.2) on real-world open-source patch scenarios. Thus, our prototype improves the state-of-the-art and may be used right away.

While *mpatch* presents an improvement over current patchers, some patches can still not be applied automatically. The main reason is that the prototype's implementation of the match operator cannot handle certain differences between source and target well, such as renaming or moving files, or reordering text. Furthermore, the filter heuristic represents a balance between precision and recall, and it is sometimes not possible to distinguish required from undesired changes. If the filter is too strict, *mpatch* will reject too many required changes. If it is too lenient, *mpatch* will apply too many incorrect changes. While there may be better matching heuristics that exhibit different trade-offs on accuracy and performance to tackle such outliers, LCS already leads to a considerable improvement over other patchers for most scenarios.

***RQ2**: Our evaluation presents strong empirical evidence that mpatch outperforms current language-agnostic patchers on a wide spectrum of patch scenarios. Using mpatch, practicioners could on average apply 44% more patches automatically, requiring no manual fixing, than the best current language-agnostic patcher.*

***RQ3**: Potential impact of improving language-agnostic patching with mpatch.* Finding that *mpatch* improves patching for individual patching scenarios, we also gauge its potential impact for developers, in general. To this end, we first consider the prominence of patching in public repositories, and second, the heavy reliance of some projects on patching.

While mining cherry picks from repositories, we were only able to identify cherry picks in 696 out of 5,000 repositories, but we may have missed patches that were applied with other tools than *Git cherry-pick*. As we only selected cherry picks with dedicated, optionally-generated commit messages (cf. Section 4.1), it is likely that we missed an unknown number of cherry picks without this message, and patch scenarios that were created with other patchers or manual copy-and-pasting. Despite this limitation, we were able to identify about 423k patches, with about 101k of them being complex. This corresponds to about 144 complex cherry picks per project with cherry picks. Of these 101k complex patches, *mpatch* is able to automatically apply about 17k more patches than the best current patcher. For each of these patches, manual effort is saved in addition to requiring fewer fixes in most cases.

We also investigated the potential impact of *mpatch* in the Top-5 projects with the highest number of cherry picks (cf. Table 4). Here, we observe that patching is a central aspect for three of these five projects, as more than 17% of all commits in their history were patched (i.e., cherry picked). For these projects, *mpatch* could lead to a substantial reduction in manual effort that is caused by fixing patches. The most extreme case we found is in ceph: 22% of their commits are patches. In ceph, *mpatch* could have applied 1,800 more patches automatically than *Git cherry-pick*, while requiring only half the number of fixes.

***RQ3**: Our evaluation shows that mpatch could have a large positive impact on patching in many different projects and domains, especially projects that heavily utilize patching.*

## 6.2 Threats to Validity

*Internal Validity.* As with any software, bugs in our evaluation setup might lead to incorrect conclusions. To ensure the correctness of *mpatch*, we implemented unit and integration tests that check its match, filter, and apply phase in various patching scenarios. Whenever applicable, we also integrated implementations of trusted libraries into our prototype, e.g., for the LCS matcher. We further reviewed edge cases and anomalies to double-check our complete evaluation work flow.

In our evaluation we calculate various metrics that are influenced by the outcomes of Unix `diff`; we use it to assess the difference of the expected patch target and the achieved patch target. Such a diff is intrinsically heuristic, i.e., there are many different but valid diffs of two files. *GNU patch*, *Git apply*, and our *mpatch* use a diff as input, and thus may be biased by this.

All patch scenarios in our evaluation were originally performed using *Git cherry-pick*, which introduces two biases in its favor. First, *Git cherry-pick*'s patch results are often more similar to our evaluation's ground truth, because the ground truth was derived from *Git cherry-pick*: each of our patch scenarios is the result of a developer using *Git cherry-pick*, and then fixing its result, i.e., resolving merge conflicts. In each patch scenario, a developer thus started to resolve conflicts after they were presented the *Git cherry-pick* results of our evaluation. Second, our dataset may miss cases where *Git cherry-pick* performed exceptionally poor. Such cases could be either *Git cherry-pick* crashing or creating too many merge conflicts to be managable. Instead of attempting to deal with these scenarios and commit their result, these scenarios may have just been skipped or handled with another tool. It is thus noteable that *mpatch* still performed better than *Git cherry-pick*.

We evaluated all patchers in their default configuration. Each patcher offers various fine-tuning options for specific tasks, and using these configurations may yield different outcomes. However, we chose to evaluate only the most straightforward, and likely most commonly used default configurations. We argue that this approach provides a good impression of a patcher's effectiveness.

*External Validity.* Our dataset is limited to public projects on GitHub and our observations thus might not be generalizable to closed-source projects. However, many professional developers participate in projects on GitHub and our dataset of diverse projects should cover many different programming practices. Similarly, our results may not generalize to less popular projects. However, a better patching tool would have less impact on these, anyway.

In our evaluation, we only consider patch scenarios that stem from cherry picks and thus *Git cherry-pick*. Currently, our knowledge of patch scenarios from other tools is severely limited, and we have no reliable way of collecting them, automatically. However, our dataset of patches is huge and covers various domains, broadening our knowledge at least for cherry picks in open source.

# 7 Related Work

## 7.1 Techniques Related to Patching

*Language-specific Patching.* Almost all language-specific patchers were designed for patch backporting. Patch backporting creates a patch from changes applied to a newer version of a program and transforms this patch to fit an older version. Most backporting techniques focus on backporting patches for the Linux kernel [35], some specialize on Linux device drivers only [31, 38], while others are applicable for code of a specific language; e.g., C code [35, 45] or PHP [36]. Harnessing the syntax, semantics, control flow, or dependencies of their application domain, they (e.g., `FixMorph` [35] and `TSBPORT` [45]) likely outperform language-agnostic patchers. However, this effectiveness comes at the price of limited applicability, specifically to the chosen file types and languages. For this reason, language-specific patchers are not widely adopted. The patches in our datasets contain changes to more than 1,000 different file types and more than 40% of the patched files are not written in any of the project languages (e.g., txt, md, xml, json), which hinders the application of a language-specific patcher. We consider a direct comparison with language-specific patching out of the scope of

this paper as backporting techniques are complementary to *mpatch*: While *mpatch* can be used for language-agnostic patching spanning hundreds of different languages and file types, language-specific patchers could be preferred for their specialized use cases focusing changes to files written in supported languages.

*LLM-based patching.* Recently, Pan et al. [28] proposed the first LLM-based patching approach called $PP_{AT}HF$ for applying patches between hard project forks. By using an LLM, their approach is in principle language-agnostic. However, it was specifically designed for the patching of functions (aka. procedures), and was only evaluated on patches to C functions in (Neo)vim. Thus, it is unclear how the applicability of their approach generalizes to other programming languages and whether it can be used for other artifacts (e.g., configuration files, or documentation) and larger patches.

*Patching in Variant-Rich Systems.* In a prior work, we investigated the potential to automate patching for clone-and-own variants (i.e., diverged versions) [32]. We simulated diverging versions of the Unix suite BusyBox and observed that *GNU patch* achieved a high precision and recall of 0.92 and 0.93. While we report a similar precision for *GNU patch* in this paper, we found that its recall does not generalize to patch scenarios in public projects, as the average recall is only 0.80. Research on patch propagation [26], patch mutation [4], patch filtering [17], or differencing [5] for configurable software (i.e., software product lines) assumes the existence of explicit documentation on (1) the available software versions or variants, and (2) the relation of source code to configuration options (e.g., via C preprocessor annotations). Techniques that rely on an *integrated platform* (e.g., [1, 2, 2, 11, 19–22, 34, 37, 42]) propagate changes to software variants automatically but require a single representation of all software variants similar to software product lines. While such methods are effective when explicit knowledge on variability is present, they are neither applicable as language-agnostic patchers nor account for cases without this explicit knowledge.

## 7.2 Studies on Patching Practices

Businge et al. [7] conducted a study on patching between forks in Android, .NET, and JavaScript systems. They found that the rate of patching between forks is overall low and that most patches are applied to forks as pull requests (i.e., merges). They also investigated patching between two forks and observed that *Git cherry-pick* is seldom used (9% of Android, 0.9% of .NET, and 2.5% of JavaScript). By contrast, we found cherry-picks in about 14% of the projects in our dataset. We suspect that this discrepancy is due to their more conservative method of considering only trivial cherry-picks. Ramkisoen et al. [30] investigated the "patch technical lag" in divergent forks: They found that it takes 27 weeks on average until a bug fix is propagated to a fork that requires it. Jang et al. [13] investigated vulnerability patching over time. They found that the majority of clones remain unpatched after one year, and that some clones remain unpatched even longer. All of these empirical findings suggest a need for better tool support that helps with identifying and applying patches. Li et al. [18] investigated patch porting strategies among different Linux distributions. They found a variety of porting strategies and that patch porting appears to be a trade-off between introducing too many patch-related bugs and missing too many important patches. Regarding the complexity of patching, Shariffdeen et al. [35] investigated the typical number of patches that are backported and the time required to backport them. They found that many patches are backported to Linux kernel versions (about 8% per Linux version) and that backporting requires more than 20 days for 80% of patches. Furthermore, when analyzing patches they observed that only 23% of backported patches were trivial. This suggests that patch backporting is more complex than patching in general, as we observed that more than 75% of patches in our dataset were trivial.

# 8 Conclusion

In this paper, we investigated the prominence of patching in public repositories and the effectiveness of patchers in complex patch scenarios. In complex scenarios, the source and target version of a patch diverged and exhibit differences that patchers have to handle. For these, we observed that patchers apply patches with high precision, but low recall as current patchers struggle to identify the correct change locations. This causes an overall low rate of automation of only 16% to 41% of the investigated patch scenarios.

To address these shortcomings, we introduced *mpatch*, a novel language-agnostic patching technique with a match-based strategy. *mpatch* achieves a considerably higher automation rate than the best current patcher in the studied patch scenarios, and can apply 44% more patches automatically without mistakes. We investigated the potential impact of *mpatch* on patching in practice. To this end, we mined a dataset of over 400,000 scenarios from 5,000 open source repositories on GitHub, of which over 100,000 were identified as complex. Of these complex scenarios, *mpatch* can correctly and automatically apply 17k patches more than the best current patcher. This directly impacts projects that heavily rely on patching, such as ceph, in which we found more than 7,000 complex patches, of which *mpatch* applies 1,800 more patches automatically.

In conclusion, our work shows that language-agnostic patching is a challenging problem in software maintenance and evolution. Particularly, when the source and target of a patch diverge, current language-agnostic patchers fall short. Consequently, our work paves the way for more efficient maintenance and evolution of complex software projects, for instance, by integrating *mpatch* into version control systems. Additionally, our large dataset of complex patches can be used to evaluate the effectiveness of novel patching techniques, or to study patching practices in public repositories. Finally, our design and implementation of a match-based patcher demonstrated superior effectiveness compared to a family of tools that have evolved and matured over almost 40 years. This underscores the importance of re-evaluating even the most established methods and techniques in our more than vibrant research area of software engineering.

# Acknowledgments

This work is part of the VariantSync project and was supported by the Swiss National Science Foundation under grant 222903, and by the German Research Foundation under grant TH 2387/1-2.

# References

[1] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf H.

Reussner. 2022. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering (EMSE)* 27, 5 (2022), 101. doi:10.1007/s10664-021-10097-z

[2] Sofia Ananieva, Thomas Kühn, and Ralf Reussner. 2022. Preserving Consistency of Interrelated Models During View-Based Evolution of Variable Systems. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 148–163. doi:10.1145/3564719.3568685

[3] Fabian Beuke. 2024. GitHut 2.0 – A Small Place to Discover Languages in GitHub. Website: https://madnight.github.io/githut/#/stars/2024/1. Accessed: 2024-06-01.

[4] Paul Maximilian Bittner, Alexander Schultheiß, Sandra Greiner, Benjamin Moosherr, Sebastian Krieter, Christof Tinnes, Timo Kehrer, and Thomas Thüm. 2023. Views on Edits to Variational Software. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 141–152. doi:10.1145/3579027.3608985

[5] Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Timo Kehrer, and Thomas Thüm. 2024. Variability-Aware Differencing with DiffDetective. In *Companion Proc. Int'l Conference on the Foundations of Software Engineering (FSE Companion)*. ACM, 632–636. doi:10.1145/3663529.3663813

[6] Alexander Boll, Yael Van Dok, Manuel Ohrndorf, Alexander Schultheiß, and Timo Kehrer. 2024. Towards Semi-Automated Merge Conflict Resolution: Is It Easier Than We Expected?. In *Proc. Int'l Conf. on Evaluation Assessment in Software Engineering (EASE)*. ACM, 282–292. doi:10.1145/3661167.3661197

[7] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. 2022. Reuse and Maintenance Practices Among Divergent Forks in Three Software Ecosystems. *Empirical Software Engineering (EMSE)* 27, 2 (2022), 54. doi:10.1007/S10664-021-10078-2

[8] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)* 30, 2 (1998), 232–282. doi:10.1145/280277.280280

[9] Edward E Cureton. 1956. Rank-Biserial Correlation. *Psychometrika* 21, 3 (1956), 287–290.

[10] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. 2005. Impact of Software Engineering Research on the Practice of Software Configuration Management. *Trans. on Software Engineering and Methodology (TOSEM)* 14, 4 (2005), 383–430. doi:10.1145/1101815.1101817

[11] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 665–668. doi:10.1109/ICSE.2015.218

[12] Stack Exchange Inc. 2023. Beyond Git: The Other Version Control Systems Developers Use. Website: https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/. Accessed: 2024-07-01.

[13] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proc. IEEE Symposium on Security and Privacy (SP)*. IEEE, 48–62. doi:10.1109/SP.2012.13

[14] Jesper Juhl. 2016. Applying Patches To The Linux Kernel. Website: https://www.kernel.org/doc/html/v4.11/process/applying-patches.html. Accessed: 2024-07-01.

[15] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. 2012. Adaptability of Model Comparison Tools. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 306–309. doi:10.1145/2351676.2351731

[16] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-Preserving Edit Scripts in Model Versioning. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 191–201. doi:10.1109/ASE.2013.6693079

[17] Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. 2024. Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)* (Dommeldange, Luxembourg). ACM, New York, NY, USA, 12–23. doi:10.1145/3646548.3672585

[18] Xingyu Li, Zheng Zhang, Zhiyun Qian, Trent Jaeger, and Chengyu Song. 2024. An Investigation of Patch Porting Practices of the Linux Kernel Ecosystem. In *Proc. Working Conf. on Mining Software Repositories (MSR)* (Lisbon, Portugal). ACM, New York, NY, USA, 63–74. doi:10.1145/3643991.3644902

[19] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 49–62. doi:10.1145/3136040.3136054

[20] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and Systems Modeling (SoSyM)* 16, 4 (2017), 1179–1199. doi:10.1007/s10270-015-0512-y

[21] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of Variation Control Systems. *J. Systems and Software (JSS)* 171 (2021), 110796. doi:10.1016/j.jss.2020.110796

[22] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1658–1670. doi:10.1109/ICSE43902.2021.00147

[23] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Sydit: Creating and Applying a Program Transformation From an Example. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 440–443. doi:10.1145/2025113.2025185

[24] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations From an Example. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 329–342. doi:10.1145/1993316.1993537

[25] Tom Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Trans. on Software Engineering (TSE)* 28, 5 (2002), 449–462. doi:10.1109/TSE.2002.1000449

[26] Gabriela Karoline Michelon, Wesley K. G. Assunção, Paul Grünbacher, and Alexander Egyed. 2023. Analysis and Propagation of Feature Revisions in Preprocessor-based Software Product Lines. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*, Tao Zhang, Xin Xia, and Nicole Novielli (Eds.). IEEE, 284–295. doi:10.1109/SANER56733.2023.00035

[27] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. doi:10.1007/BF01840446

[28] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2024. Automating Zero-Shot Patch Porting for Hard Forks. In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*, Maria Christakis and Michael Pradel (Eds.). ACM, New York, NY, USA, 363–375. doi:10.1145/3650212.3652134

[29] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. 2004. *Version Control With Subversion*. O'Reilly Media, Inc.

[30] Poedjadevie Kadjel Ramkisoen, John Businge, Brent van Bladel, Alexandre Decan, Serge Demeyer, Coen De Roover, and Foutse Khomh. 2022. PaReco: Patched Clones and Missed Patches Among the Divergent Variants of a Software Family. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 646–658. doi:10.1145/3540250.3549112

[31] Luis R. Rodriguez and Julia Lawall. 2015. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In *Proc. Europ. Dependable Computing Conf. (EDCC)*. IEEE, 132–143. doi:10.1109/EDCC.2015.23

[32] Alexander Schultheiß, Paul Maximilian Bittner, Thomas Thüm, and Timo Kehrer. 2022. Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 269–280. doi:10.1109/ICSME55016.2022.00032

[33] Alexander Schultheiß, Alexander Boll, Paul Maximilian Bittner, Sandra Greiner, Thomas Thüm, and Timo Kehrer. 2025. The Reproduction Package of this Paper. Website: https://doi.org/10.5281/zenodo.16920961.

[34] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 822–827. doi:10.1145/2970276.2970288

[35] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated Patch Backporting in Linux (Experience Paper). In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*. ACM, 633–645. doi:10.1145/3460319.3464821

[36] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches. In *Proc. USENIX Security Symposium (USS)*. USENIX Association, 1993–2010.

[37] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333. doi:10.1109/ICSME.2016.88

[38] Ferdian Thung, Xuan-Bach Dinh Le, David Lo, and Julia Lawall. 2016. Recommending Code Changes for Automatic Backporting of Linux Device Drivers. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 222–232. doi:10.1109/ICSME.2016.71

[39] Walter F. Tichy. 1982. Design, Implementation, and Evaluation of a Revision Control System. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 58–67.

[40] Linus Torvalds, Junio C Hamano, et al. 2023. git-apply. Website: https://git-scm.com/docs/git-apply. Accessed: 2024-07-01.

[41] Linus Torvalds, Junio C Hamano, et al. 2023. git-cherry-pick. Website: https://git-scm.com/docs/git-cherry-pick. Accessed: 2024-07-01.

[42] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 29–38. doi:10.1145/2658761.2658766

[43] Larry Wall, Paul Eggert, Wayne Davison, David MacKenzie, and Andreas Grünbacher. 2009. GNU patch. Website: https://savannah.gnu.org/projects/patch/. Accessed: 2024-07-01.

[44] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. doi:10.2307/3001968

[45] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. 2023. Enhancing OSS Patch Backporting with Semantics. In *Proc. SIGSAC Conf. on Computer and Communications Security (CCS)*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2366–2380. doi:10.1145/3576915.3623188